JavaScript - Perusteet 2

- Tässä luvussa esitellään pääsääntöisesti monia keskeisimpiä ES6-version mukana tulleita uusia ominaisuuksia.

> Tämän materiaalin muissa luvuissa saatetaan käyttää tekniikoita, jotka ovat mahdollisia myös aiemmille ES-versioille. Tällaisia ovat esim. var-avainsanalla määritellyt muuttujat. Opintojakson harjoitustehtävien, osaamistestien ja harjoitustyön ratkaisuissa voi käyttää myös näitä aiempien ES-versioiden tukemia tekniikoita, ellei erikseen ole vaadittu ES6-toiminnallisuutta.

let

- Muuttujan määrittely
- Uutta ES6-syntaksissa, pyri käyttämään var-määrittelyn sijaan
- Ohjelmalohkon (block) sisällä määriteltynä näkyy VAIN lohkon sisälle

Esimerkki 0301

```
let ika = 35;
console.log(ika);
if (true) {
let ika = 53;
console.log(ika);
}
/*
Tulostaa:
35
53
*/
```

const

- Vakion määrittely
- Uutta ES6-syntaksissa, pyri käyttämään var-määrittelyn sijaan

- Vakiolle ei voi sijoittaa uutta arvoa

Esimerkki 0302

```
const IKA = 35;
IKA = 53; // Type Error ....
```

Taulukon ja objektin **ominaisuutta voi** muuttaa, mutta vakio-objektille EI voi sijoittaa kokonaan uutta objektia. (uusi muistiosoite)

Esimerkki 0303

```
const LUVUT = [35, 53];
console.log(LUVUT); // Tulostaa: [35, 53]

LUVUT.push(175);
console.log(LUVUT); // Tulostaa: [35, 53, 175]


const OPE = {
        ika: 35
};

OPE.ika = 53
console.log(OPE.ika); // Tulostaa: 53
```

Olio-ohjelmointi ja JavaScript [1]

- JavaScriptin oliomalli poikkeaa merkittävistä muista kielistä (mm. Java ja C++)
- JavaScriptissä ei ole varsinaista luokka-käsitettä vaan kaikki oliot ovat samaa object-tyyppiä.
- Myös ES6:n luokat ovat pääosin vain syntaktista sokeria
- Javacriptissä olio (object) on vain kokoelma ominaisuuksia (properties)

```
let henkilo = {nimi: 'Arska', ika: 49}
```

- Viittaaminen hakasulkumerkinnällä

```
henkilo['nimi']
```

- Viittaaminen pistemerkinnällä

```
henkilo.nimi
```

- Oliolle voidaan lisätä ominaisuuksia

```
henkilo.hetu = '336666-012Q';
```

Olio-ohjelmointi ja JavaScript [2]

- Taulukko on erikoistapaus, jonka alkioihin voi viitata **vain** hakasulkumerkinnällä.
- Olion ominaisuutena voi olla myös funktio, jolloin puhutaan metodista
- Metodi voi viittaa olioon itseensä **this**-avainsanalla

Olio-ohjelmointi ja JavaScript [3]

- Käytetään valmiita JavaScript-olioita (kuten Math, Date)
- Käytetään HTML-oliota (kuten document, window, form, image jne.)
- Käytetään itse kehitettyjä olioita

Olio-ohjelmointi ja JavaScript [4]

- Nykyisessä (ES6) JavaScriptissä ei ole perinteistä tapaa luoda luokkia public-, protected- ja private-määritellyin ominaisuuksin
- Muisti oliolle varataan new-operaattorilla
- **ES5**: Funktio toimii luotavan olion konstruktorina ja määrittelynä (kuten luokan määrittely muissa kieleissä)
- **ES6**: Konstruktori määrillään constructor-avainsanalla

Anonyymi funktio

- JavaScript-ohjelmointia voidaan toteuttaa funktionaalisesti.
- Funktioita voidaan sijoittaa muuttujiin ja kutsua muuttujien kautta.
- Funktionaalisessa ohjelmoinnissa on tärkeää, että funktio ei muuta parametriensa arvoja vaan palauttaa (tarvittaessa) arvon return-lauseessa.

```
function Car(malli) {
 ...
 this.getColor = function(color) { // Anonyymi funktio
   return carColor;
}
```

Esimerkki 0304 - Olion luonti ja käyttö

```
function Car(malli) {
  // public property
  this.model  = malli;

  // private variable (voi käyttää vain tässä scopessa)
  var carColor = "Red";
 // public setter and getter  methods
 this.getColor = function(){
  return carColor;

 }
 this.setColor = function(color) {
   carColor = color;
 }
}

// Luodaan uusi auto
var ford = new Car("Ford");
console.log(ford.model); // Ford
console.log(ford.getColor()); // Red
ford.setColor("Blue");
console.log(ford.getColor()); // Blue

// Testataan private-variable
console.log(ford.carColor); // Määrittelemätön

// Testataan carColor, ei global scope
console.log(carColor); // Määrittelemätön
```

Prototyyppi

- JavaScriptissä voit lisätä olemassa olevaan olioon määrittelynsä jälkeen ominaisuuksia (muuttujia ja funktioita)

```
ford.nopeus = 165;
ford.getNopeus = function(){return nopeus;};
```

- prototype-ominaisuus sallii lisäksi lisätä ominaisuuksia olemassaolevaan prototyyppiin eli kaikkiin prototyyppiä käyttäviin olioihin. Tällöin on käytettävä prototype-avainsanaa

```
Car.prototype.pa = 'diesel';
Car.prototype.getPa = function(){return pa;};
```

- Jatkossa prototyyppiin lisättyjä ominaisuuksia voi käyttää kuten alkuperäisestikin määriteltyjä ominaisuuksia

```
console.log(ford.getPa()); // diesel
console.log(opel.getPa()); // diesel
console.log(toyota.getPa()); // diesel
```

Callback-funktiot

- Callback-funktio on funktio, joka annetaan toiselle funktiolle kutsussa argumenttina.
- Toisen funktion suorituksen aikana on sitten mahdollista kutsua annettua callback-funktiota haluttun käsittelyn aikaansaamiseksi.
- Callback-funktioita käytetään paljon esim. tapahtumankäsittelijöissä

Esimerkki 0305

- Synkroninen callback, joka suoritetaan välittömästi

```
function fn1(name) {
  alert('Terve ' + name);
}

function fn2(name) {
  document.write('Terve ' + name);
}

function lueSyote(callbackfunktio) {
  var name = prompt('Kirjota nimesi');
  callbackfunktio(name); // Kutsutaan kutsussa annettua funktiota
}

lueSyote(fn1); // Tässä valitaan syötteen käsittelijäfunktio
               // Kokeile myös kutsulla: lueSyote(fn2)
```

Asynkroninen callback

- Callcack-funktioita käytetään paljon asynkronisesti, jolloin callback-kutsu suoritetaan vasta tietyn asynkronisen toiminnan valmistuttua.
- Esimerkiksi jos callback-funktion tehtävänä on renderöidä palvelimen tietokannasta haettavaa dataa selainikkunaan käyttäjän syötteen perusteella, niin tämä renderöinti voidaan aloittaa tietenkin vasta sitten kun kaikki tarvittava data on saatu haettua.

Esimerkki 0306

- Kirjastofunktion `setTimeout` kutsun ensimmäinen argumentti määrää sen callback-funktion, jota kutsutaan toisena argumenttina annetun millisekuntien kulumisen jälkeen

```
function fn() {
   console.log("Haloo");
}

//Tulostaa "Haloo" (1 sec kuluttua)
setTimeout(fn, 1000);
```

Lambdat ~ arrow functions

- Uutta ES6-syntaksissa, yksi merkittävistä muutoksista
- Lambda-merkinnät ~ arrow functions ~ nuolifunktiot ~ fat arrow functions
- Tapa merkitä anonyymi funktio kompaktisti ja luettavasti
- Lambda-merkityllä funktiolla EI ole omaa this-viittausta

Esimerkki 0307 - Lambda-merkinnät tiiviisti

- Huomaa erityisesti saman toiminnalliisen lihavoidut rivit, joista jälkimmäinen tiiviimpi.

```
// Perinteinen nimetty funktio
function fn(a) { return a; }

// Perinteinen anonyymi funktio
let afn = function (a) { return a; }

// Lambda-merkitty anonyymi funktio
let lfn1 = (a) => a;

// Lambda: Jos yksi parametri, sulut saa jättää pois
let lfn2 = a => a;

console.log(fn("From fn"));     // "From fn"
console.log(afn("From afn"));   // "From afn"
console.log(lfn1("From lfn1")); // "From lfn1"
console.log(lfn2("From lfn2")); // "From lfn2"


// Lambda: Ei parametreja, tyhjät sulut
let lfn3 = () => {console.log("From lfn3");};
lfn3(); // "From lfn3"

// Lambda: Useat parametrit normaalisti
let lfn4 = (a, b) => a + b;
console.log(lfn4(-1, 5)); // Tulostaa: 4

// lambda-merkitty funktiokin voi
// suorittaa useita lauseita
let lfn5 = () => {
  console.log("From lfn5 row 1");
  console.log("From lfn5 row 2");
```

```
};

lfn5(); // "From lfn5 row 1"
        // "From lfn5 row 2"
```

Lambda callback-funktiona

- Lambda-merkityt funktiot sopivat lyhyinä callback-funktioiksi

```
//Tulostaa "Haloo" (1 sec kuluttua)
setTimeout(() => console.log("Haloo"), 1000);
```

Esimerkki 0308 - Lambdat ja this-viittaus

Huomaa vaihtaa kommenttia kahdella viimeisellä rivillä kutsuaksesi joko `anofn()`-funktiota tai `lambdafn()`-funktiota!

```
//HTML:
<body>
<button>Klikkaa</button>
</body>

//JS:
var btn = document.querySelector('button');

function anofn() {
  return console.log(this);
}

var lambdafn = () => console.log(this);

btn.addEventListener('click', anofn);
//btn.addEventListener('click', lambdafn);
```

- Anonyymiä funktiota `anofn()` kutsuttaessa buttonia klikaten konsolillle tulostetaan HTMLButtonElement-objekti [object HTMLButtonElement] eli se objekti, joka kutsui funktiota! => Eli this sisältää viittauksen **kutsuvaan** objektiin!
- Lambda-merkittyä funktiota `lambdafn()` kutsuttaessa buttonia klikaten konsolillle tulostetaan Window-objekti [object Window] eli se objekti, jossa lambda-merkitty funktio on määritelty => Eli this sisältää viittauksen objektiin, jossa lambda-merkitty funktio on määritelty! Tämä on se mitä useimmin halutaan

Esimerkki 0309 - this-ongelma anonyymillä funktiolla

- Edellisestä seuraa ongelma perinteisellä anonyymillä funktiolla, joka pitää ratkaista ylimääräisellä self-muuttujalla

- Ilman nähtyä järjestelyä setInterval-funktion ensimmäisen parametrin (callback-funktio) viittaukset Counter-olion number-ominaisuuteen eivät onnistuisi. => Tästä seuraa sekavaa ohjelmakoodia
- Ohjelman toiminta: setInterval-funktio kutsuu callback-funktiona anonyymiä funktiota puolen sekunnin välein ja pävittää laskurin kasvavaa lukemaa H3-elementin sisällöksi

```
//HTML:
<body>
  <h3 id=count>0</h3>
</body>

//JS:
function Counter() {
  this.number = 0;
  let self = this;
  setInterval( function() {
    self.number++;
    document.getElementById("count").innerHTML = self.number;
  }, 500);
}

let counter = new  Counter();
```

Esimerkki 0310 - this-ongelman välttäminen lambda-merkinnällä

- this-viittausta voi käyttää lambda-merkityssä callback-funktiossa, koska se viittaa siihen kontekstiin, jossa se on määritelty eli Counter-objektiin
- Toiminta on sama kuin 0307:ssa, mutta selkeämmällä syntaksilla

```
//HTML:
<body>
  <h3 id=count>0</h3>
</body>

//JS:
function Counter() {
  this.number = 0;

  setInterval(() => {
    this.number++;
    document.getElementById("count").innerHTML = this.number;
  }, 500);

}

let counter = new  Counter();
```

Oletusparametrit

- Uutta ES6-syntaksissa
- Esitellään esimerkkien avulla. Kokeile itse ja tutki toiminta.

Esimerkki 0311

```
function fn1(n1, n2 = 2) {
  return n1 == n2;
}

console.log(fn1(3)); // false
console.log(fn1(2)); // true

// ----------------------------------------

function fn2(n1, n2 = n1) {
  console.log(n1); // 4
  console.log(n2); // 5
  return n1 == n2;
}

console.log(fn2(4, 5)); // false, koska sijoitus n2 = n1
                        // on vain oletusarvo

// ----------------------------------------

function fn3(n1 = n2, n2 = 42) {
  return n1 == n2;
}

console.log(fn3()); // Error, koska parametrit käsitellään
                    // järjestyksessä n2 not-defined, kun sitä
                    // yritetään käyttää ekan kerran
```

Olioliteraalit

- Uutta ES6-syntaksissa

Esimerkki 0312

- Muuttujista saadaan oletusarvot kentille, kuten alla **ika**-muuttujan arvo

```
let nimi = 'Sepi';
let ika = 53;

let olio = {
 nimi : 'Ari',
 ika,
 fn() {
  console.log(this.nimi + ', ' + this.ika)
}
}

console.log(olio);  /* Tulostaa:
                       [object Object] {
                         ika: 53,
                         nimi: "Ari"
                       }*/

olio.fn(); // "Ari, 53"
```

- Voidaan käyttää myös lyhennettyä merkintää, jolloin ominaisuudet nimetään muuttujien mukaan

```
let nimi = 'Ari';
let ika = 53;

let olio = {nimi, ika};
console.log(olio.nimi); // "Ari"
```

Olioliteraalit - Dynaamiset kentät

- Uutta ES6-syntaksissa
- olion kenttien nimet voidaan muodostaa dynaamisesti hakasulkumerkinnällä []

```
let counter = 0;
let getId = () => ++counter;

let olio1 = {
              ['id_' + getId()]: counter
            };

let olio2 = {
              ['id_' + getId()]: counter
            };


console.log(olio1); // {id_1: 1}
console.log(olio2.id_2); // "2"
```

Rest-operaattori: ...taulukko

- Uutta ES6-syntaksissa
- Rest-operaattori **funktion parametrina** muuntaa kutsussa annetun listan taulukoksi

Esimerkki 0313

```
function summaa(...luvut) { // Muuntaa annetun listan tauluksi
  console.log(luvut);
  let tulos = 0;
  for(let i = 0; i < luvut.length; i++) {
    tulos += luvut[i];
  }
  return tulos;
}

console.log(summaa(10,20,30)); // Lista
```

```
// Tulostaa:
// [10, 20, 30]
// 60
```

Spread-operaattori: ...taulukko

- Uutta ES6-syntaksissa
- Spread-operaattori **funktion kutsussa** muuntaa kutsussa annetun taulukon listaksi
- Tarpeellinen vaikkapa kirjastofunktion vaatiessa listaa silloin kun syöttöarvot ovat taulukossa

Esimerkki 0314

```
let luvut  = [1,2,3];
console.log(...luvut); // 1
                       // 2
                       // 3
console.log(luvut);  // [1,2,3]

// Math-olion max-metodi vaatii listaa, mutta
// arvot ovat taulussa, jolloin taulu
// voidaan muuntaa listaksi tuloksen saamiseksi:
console.log(Math.max(...luvut)); // 3
```

Template-literaalit

- Uutta ES6-syntaksissa
- "tavallinen merkkijono parilla lisäominaisuudella"

Esimerkki 0315

- Voidaan määritellä monirivinen rivinvaihdot sisältävä merkkijono `gravis aksentti` -merkeillä ympäröitynä. Toimii samankaltaisesti kuin *"Here Document"*-syntaksi useissa kielissä
- Gravis aksentti -merkki eli "takakenohipsu" saadaan BackSpace-näppäimen vasemmalta puolelta nk. DeadKey-näppäilyllä eli painamalla ensin yhtäaikaa SHIFT+GravisAksentti, nostamalla kummatkin näppäimet ylös ja painamalla Space-näppäintä. Helppoa, eikö totta!
- Huomaa gravis aksentti -merkit toisen rivin lopussa ja neljännen alussa

```
let nimi = 'Ari';
let loitsu = `
ES6 rules ok
`;
console.log(loitsu);  "ES6 rules ok"
```

Esimerkki 0316

- Template-literaalien muodostamisessa voi käyttää osana myös muuttujien arvoja `${}`-syntaksilla. Esimerkki valaisee asiaa:

```
let nimi = 'Arska';
let loitsu = `
Helou, I'm ${nimi}
Helou, I'm ${nimi + ' - Raaka Arska!'}
Helou, I'm \${nimi + ' - Raaka Arska!'}
`;
console.log(loitsu);

/* Viimeinen lause tulostaa:
"
Helou, I'm Arska
Helou, I'm Arska - Raaka Arska!
Helou, I'm ${nimi + ' - Raaka Arska!'}
"
*/
```

Taulukoiden ja objektien purkaminen muuttujiin

- Uutta ES6-syntaksissa
- Tällä syntaksilla voidan taulukon arvojoukko tai objektin kenttien arvot joko kokonaisuudessaan tai osittain kopioida halutulle muuttujajoukolle.

Esimerkki 0317

```
let luvut = [1, 2, 3];

let [n1, n2, n3] = luvut;
console.log(n1); // 1
console.log(n2); // 2
console.log(n3); // 3
```

Esimerkki 0318

```
let luvut = [1,2,3]
let [a,...b] = luvut;
```

```
console.log(b); // [2, 3]
```

Esimerkki 0319

- Myös olion ominaisuudet voidaan sijoittaa tavallisille muuttujille

```
let olio = {
  nimi: 'Ari',
  ika: 53
};

 // nimi-kentän uudelleennimäminen -> name

let { nimi: name, ika } = olio; // name on uusi aliasnimi

console.log(name); // "Ari"
console.log(ika); // "53"
console.log(nimi); // "error" (ei määritelty)
```

Esimerkki 0320

```
let a = {
  nimi: 'Ari',
  ika: 53,
  fn: function() {
    console.log('Haloo');
  }

};

let {nimi, , fn} = a;
fn(); // Error, koska objektin kentillä EI järjestystä
```

- Lisää esimerkkejä: MDN - Destructuring assignment

Luokan määrittely class-määreellä

- Uutta ES6-syntaksissa
- ES6:n luokat ovat pääosin vain syntaktista sokeria ja ne ainoastaan ylikirjoittavat vanhan prototyyppipohjaisen olioiden määrittelytavan. Ne ovat täysin yhteensopivia vanhan koodin kanssa.
- Käytössä muista kielistä perinteisiä avainsanoja lähes perinteisin vaikutuksin: class, constructor, extends, super, get, set

Esimerkki 0321

```
class Person {
  constructor(name, height, weight) {
    this.name = name;
    this.height = height;
    this.weight = weight;
  }
  // Getter
  get bmi() {
    return this.calcBmi();
  }

  // Setter
  set pituus(h) {
    this.height = h;
  }

  // Metodi
  calcBmi() {
    return this.weight/(this.height/100 * this.height/100);
  }
}

const kim = new Person("Alainen Kim", 290, 90);
kim.pituus = 190;
console.log (kim.name, kim.height, kim.weight, kim.bmi);

/*
Tulostaa:
"Alainen Kim"
190
90
24.930747922437675
*/
```

Lisää esimerkkejä: MDN - Classes

Jätetty tarkoituksella tyhjäksi.